[SYSTEM-DESIGN]

Designing Data Intense Application – Chapter 8: The Trouble with Distributed Systems

Posted by CHARLES *on* 2020-04-14

<Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems>

- This chapter is a thoroughly pessimistic and depressing overview of things that may go wrong in a distributed system.
  - Networks issues
  - Clocks & timing issues

Faults and Partial Failures

- Single machine software is **deterministic**;
  - An individual computer with good software is usually either fully functional or entirely broken, but not something in between.
- In distributed systems, we are no longer operating in an idealized system model—we have no choice but to confront the messy reality of the physical world.
- The difficulty is that **partial failures** are **non-deterministic**: if you try to do anything involving multiple nodes and the network, it may sometimes work and sometimes unpredictably fail.
  - This **non-determinism** and possibility of **partial failures** is what makes distributed systems hard to work with.
- **Cloud-Computing and Super-Computing**: (C: Vertical scaling vs. Horizontal Scaling)
  - High-performance computing (HPC) vs. Cloud Computing
    - supercomputer is more like a single-node computer than a distributed system: it deals with partial failure by letting it escalate into total failure
  - If we want to make distributed systems work, we must accept the possibility of partial failure and build fault-tolerance mechanisms into the software.

- The fault handling must be part of the software design, and you (as operator of the software) need to know what behavior to expect from the software in the case of a fault.
  - important to consider a wide range of possible faults—even fairly unlikely ones.
- In distributed systems, **suspicion**, **pessimism**, and **paranoia** pay off.

Unreliable Networks

- the distributed systems we focus on in this book are **shared-nothing systems**: i.e., a bunch of machines connected by a network.
  - Why ?
    - it's comparatively cheap because it requires no special hardware,
    - it can make use of commoditized cloud computing services, and
    - it can achieve high reliability through redundancy across multiple geographically distributed datacenters.
- The internet and most internal networks in data centers (often Ethernet) are asynchronous packet networks. If you send a request and expect a response, many things could go wrong.

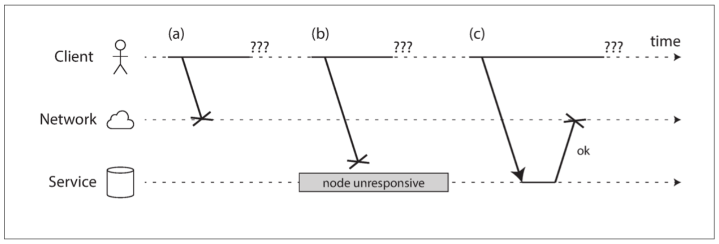

Figure 8-1. If you send a request and don't get a response, it's not possible to distinguish whether (a) the request was lost, (b) the remote node is down, or (c) the response was lost.

  - If you send a request to another node and don't receive a response, it is impossible to tell why.
    - One Solution: use Timeout.
- **Network Faults in Practice**:
  - Nevertheless, nobody is immune from network problems. Handling network faults doesn't necessarily mean tolerating them.
  - You do need to know how your software reacts to network problems and ensure that the system can recover from them.
    - "**Chaos Monkey**"(e.g. Netflix) is needed to test this "**Reliability**".
- **Detecting Faults**: (C: **Gossip** Protocol, send request/signal across each other and then broadcasting; or **ZooKeeper**)
  - Many systems need to automatically detect faulty nodes.
  - Unfortunately, the uncertainty about the network makes it difficult to tell whether a node is working or not.
- **Timeouts and Unbounded Delays**:
  - If a timeout is the only sure way of detecting a fault, then how long should the timeout be?

        example, sending an email), and another node takes over, the action may end up being performed twice.
        - When a node is declared dead, its responsibilities need to be transferred to other nodes, which places additional load on other nodes and the network.
    - **Asynchronous networks have unbounded delays** (that is, they try to deliver packets as quickly as possible, but there is no upper limit on the time it may take for a packet to arrive)
    - **Network congestion and queueing**:
        - The variability of packet delays on computer networks is most often due to queueing. (e.g. Network, CPU, VMs, TCP flow-control etc)
        - **TCP** considers a packet to be lost if it is not acknowledged within some timeout (which is calculated from observed round-trip times), and lost packets are automatically retransmitted.
        - **TCP vs. UDP**: (C: delayed message meaningless/worthless → UDP)
            - Trade-off between reliability and variability of delays: as UDP does not perform flow control and does not retransmit lost packets.
        - You can only choose timeouts experimentally.
        - Systems can continually measure response times and their variability (jitter), and automatically adjust: Phi Accrual failure detector, which is used for example in Akka and Cassandra. TCP retransmission timeouts also work similarly.
- **Synchronous vs. Asynchronous Networks**:
    - Phone call network is **synchronous**: even as data passes through several routers, it does not suffer from queueing, because the 16 bits of space for the call have already been **reserved** in the next hop of the network.
        - **bounded delay**: No queueing, the maximum end-to-end latency of the network is fixed.
    - Can we not simply make network delays predictable?
        - Ethernet and IP are packet-switched protocols, which suffer from queueing and thus unbounded delays in the network. These protocols do not have the concept of a circuit.
        - Why do datacenter networks and the internet use packet switching?
            - The answer is that they are **optimized for bursty traffic**.
    - With careful use of quality of service (QoS, prioritization and scheduling of packets) and admission control (rate-limiting senders), it is possible to emulate circuit switching on packet networks, or provide statistically bounded delay.
    - **Latency and Resource Utilization**:
        - Phone line: resource is divided up in a static way
        - Internet: shares network bandwidth dynamically
            - This approach has the downside of queueing, but the advantage is that it **maximizes utilization of the wire**.
            - A similar situation arises with CPUs.

    - Comes at the cost of reduced utilization
  - **Variable delays** in networks are not a law of nature, but simply the result of a **cost/benefit trade-off**.

Unreliable Clocks

- **Clocks and Time** are important. Applications depend on clocks in various ways.
- In a distributed system, time is a tricky business, because communication is not instantaneous.
- Each machine on the network has its own clock, which is an actual hardware device: usually a **quartz crystal oscillator**. But these devices are **not perfectly accurate**, so each machine has its own notion of time.
  - It is possible to synchronize clocks to some degree: the most commonly used mechanism is the **Network Time Protocol (NTP)**
- **Monotonic vs. Time-of-Day Clocks**: Modern computers have at least two different kinds of clocks: a **time-of-day** clock and a **monotonic** clock.
  - **Time-of-day clocks**: it returns the current date and time according to some calendar (also known as wall-clock time).
    - Time-of-day clocks are usually synchronized with NTP, which means that a timestamp from one machine (ideally) means the same as a timestamp on another machine.
  - **Monotonic clocks**: is suitable for measuring a duration (time interval), such as a timeout or a service's response time.
    - they are **guaranteed to always move forward** (whereas a time-of- day clock may jump back in time).
    - NTP may adjust the frequency at which the monotonic clock moves forward (this is known as slewing the clock) if it detects that the computer's local quartz is moving faster or slower than the NTP server.
- **Clock Synchronization and Accuracy**:
  - Monotonic clocks don't need synchronization, but time-of-day clocks need to be set according to an **NTP** server or other external time source in order to be useful.
    - Hardware clocks and NTP can be fickle beasts.
  - It is possible to achieve very good clock accuracy if you care about it sufficiently to invest significant resources.
    - E.g. using GPS receivers, the Precision Time Protocol (PTP), and careful deployment and monitoring
- **Relying on Synchronized Clocks**:
  - Robust software needs to be prepared to deal with incorrect clocks.
    - incorrect clocks easily go unnoticed.
  - If some piece of software is relying on an accurately synchronized clock, the result is more likely to be silent and subtle data loss than a dramatic crash.
  - Thus, if you use software that requires synchronized clocks, it is essential that you also carefully monitor the clock offsets between all the machines.

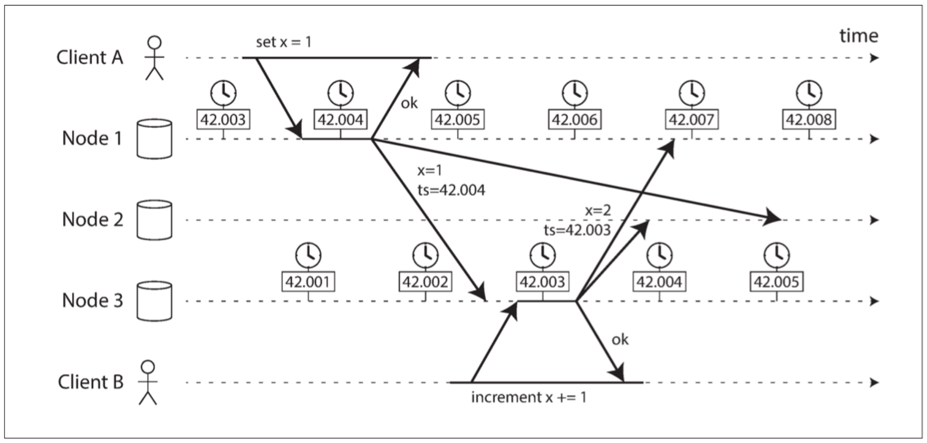

Figure 8-3. The write by client B is causally later than the write by client A, but B's write has an earlier timestamp.

- This conflict resolution strategy is called last write wins (LWW), and it is widely used in both multi-leader replication and leaderless databases such as Cassandra and Riak. (C: write to cache first?)
  - DB writes can mysteriously disappear;
  - LWW can't distinguish between writes that occurred sequentially in quick succession;
  - It is possible for two nodes to independently generate writes with the same timestamp
- For correct ordering, You would **need the clock source to be significantly more accurate than the thing you are measuring** (namely network delay).
- So-called **logical clocks**, which are based on incrementing counters rather than an oscillating quartz crystal, are a safer alternative for ordering events:
  - Logical clocks only measure the relative ordering of events (whether one event happened before or after another).
- **Clock readings have a confidence interval**:
  - Thus, it doesn't make sense to think of a clock reading as a point in time—it is more like a **range of times**, within a confidence interval.
  - Google's TrueTime API in Spanner, which explicitly reports the confidence interval on the local clock.
    - When you ask it for the current time, you get back two values: [earliest, latest]
- **Synchronized clocks for global snapshots**:
  - With lots of small, rapid transactions, creating transaction IDs in a distributed system becomes an untenable bottleneck.
  - Can we use the timestamps from synchronized time-of-day clocks as transaction IDs?
    - If we could get the synchronization good enough, they would have the right properties: later transactions have a higher timestamp.
  - Spanner needs to keep the clock uncertainty as small as possible;

  - Is it crazy to assume that a thread might be paused for so long? Unfortunately not. E.g.
    - Many programming language runtimes (such as the Java Virtual Machine) have a garbage collector (GC) that occasionally needs to stop all running threads.
    - In virtualized environments, a virtual machine can be suspended (pausing the execution of all processes and saving the contents of memory to disk) and resumed (restoring the contents of memory and continuing execution).
    - If the application performs synchronous disk access, a thread may be paused waiting for a slow disk I/O operation to complete
  - All of these occurrences can preempt the running thread at any point and resume it at some later time, without the thread even noticing.
- **Response time guarantees**:
  - In some systems, there is a specified deadline by which the software must respond; if it doesn't meet the deadline, that may cause a failure of the entire system. These are so-called **Hard real-time systems**.
  - A **real-time operating system (RTOS)** that allows processes to be scheduled with a guaranteed allocation of CPU time in specified intervals is needed;
  - For most server-side data processing systems, real-time guarantees are simply not economical or appropriate.
- **Limiting the impact of garbage collection**:
  - An emerging idea is to **treat GC pauses like brief planned outages** of a node, and to let other nodes handle requests from clients while one node is collecting its garbage.

Knowledge, Truth, and Lies

- For Distributed System: there is **no shared memory**, only **message passing** via an unreliable network with variable delays, and the systems may suffer from partial failures, unreliable clocks, and processing pauses.
  - A node in the network cannot know anything for sure—it can only make guesses based on the messages it receives (or doesn't receive) via the network.
  - A node can only find out what state another node is in (what data it has stored, whether it is correctly functioning, etc.) by exchanging messages with it.
- Although it is possible to make software well behaved in an unreliable system model, it is not straightforward to do so.
- **The Truth Is Defined by the Majority**:
  - Network with an asymmetric fault: a node is able to receive all messages sent to it, but any outgoing messages from that node are dropped or delayed.
  - In a distributed system a node cannot necessarily trust its own judgment of a situation.
  - Instead, many distributed algorithms rely on a **Quorum**, that is, voting among the nodes: decisions require some minimum number of votes from several nodes in order to reduce the dependence on any one particular node.

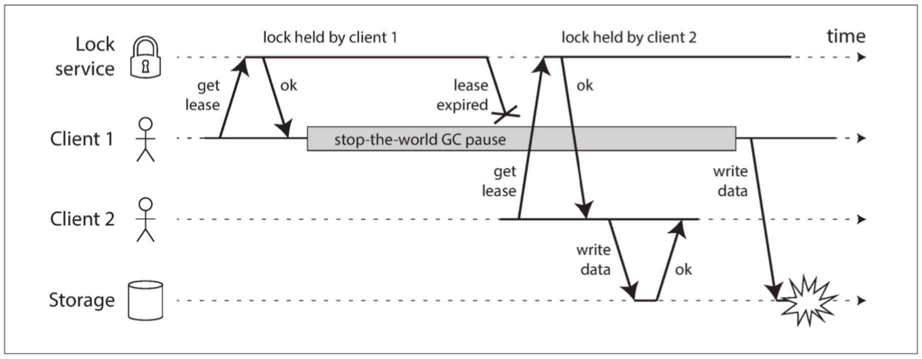

Figure 8-4. Incorrect implementation of a distributed lock: client 1 believes that it still has a valid lease, even though it has expired, and thus corrupts a file in storage.

- **Fencing tokens**:
  - we need to ensure that a node that is under a false belief of being "the chosen one" cannot disrupt the rest of the system. A fairly simple technique that achieves this goal is called **fencing**.

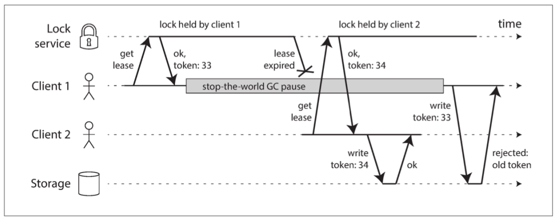

Figure 8-5. Making access to storage safe by allowing writes only in the order of increasing fencing tokens.

  - If ZooKeeper is used as a lock service, the transaction ID zxid or the node version cversion can be used as a fencing token.
- **Byzantine Faults**:
  - Distributed systems problems become much harder if there is a risk that nodes may "**lie**" (send arbitrary faulty or corrupted responses)
    - E.g. if a node may claim to have received a particular message when in fact it didn't. (Such behavior is known as a **Byzantine fault**, and the problem of reaching consensus in this untrusting environment is known as the **Byzantine Generals Problem**)
  - A system is **Byzantine fault-tolerant** if it continues to operate correctly even if some of the nodes are malfunctioning and not obeying the protocol, or if malicious attackers are interfering with the network.
  - **Weak forms of lying**:
    - it can be worth adding mechanisms to software that guard against weak forms of "lying"
      - E.g. invalid messages due to hardware issues, software bugs, and misconfiguration
    - Solutions: e.g.
      - **checksums** in the application-level protocol;
      - A publicly accessible application must carefully **sanitize** any inputs from users;

- Algorithms need to tolerate the various faults of distributed systems that we discussed in this chapter.
- Algorithms need to be written in a way that does not depend too heavily on the details of the hardware and software configuration on which they are run.

- This requires that we somehow formalize the kinds of faults that we expect to happen in a system. We do this by defining a **system model**, which is an abstraction that describes what things an algorithm may assume.
- **Timing assumptions**, three system models are in common use
  - **Synchronous** model: assumes bounded network delay, bounded process pauses, and bounded clock error.
  - **Partially synchronous** model*: means that a system behaves like a synchronous system most of the time, but it sometimes exceeds the bounds for network delay, process pauses, and clock drift.
  - **Asynchronous** model: an algorithm is not allowed to make any timing assumptions—in fact, it does not even have a clock (so it cannot use timeouts)
- **Node failures**. The three most common system models for nodes are:
  - **Crash-stop faults:**an algorithm may assume that a node can fail in only one way, namely by crashing.
  - **Crash-recovery faults***: We assume that nodes may crash at any moment, and perhaps start responding again after some unknown time.
  - **Byzantine (arbitrary) faults**: Nodes may do absolutely anything, including trying to trick and deceive other nodes.
- For modeling real systems, the partially synchronous model with crash-recovery faults is generally **the most useful model**.
- **Correctness of an algorithm**:
  - **An algorithm is correct in some system model** if it always satisfies its properties in all situations that we assume may occur in that system model.
- **Safety and Liveness**:
  - What distinguishes the two kinds of properties?
    - A giveaway is that liveness properties often include the word "**eventually**" in their definition. (And yes, you guessed it— eventual consistency is a liveness property.)
  - **Safety:** is often informally defined as **nothing bad happens**,
  - **Liveness:** as something **good eventually happens**.
  - In the example, **uniqueness** and **monotonic sequence** are safety properties, but **availability** is a liveness property.
  - **If a safety property is violated**, we can point at a particular point in time at which it was broken (e.g., if the uniqueness property was violated, we can identify the particular operation in which a duplicate fencing token was returned).

point in time (e.g., a node may have sent a request but not yet received a response),
  - but there is always hope that it may be satisfied in the future (namely by receiving a response).
- An advantage of distinguishing between **safety** and **liveness** properties is that it helps us deal with difficult system models.

- **Mapping system models to the real world**:
  - **Abstract system models** are incredibly helpful for distilling down the complexity of real systems to a manageable set of faults that we can reason about, so that we can understand the problem and try to solve it systematically.
  - Proving an algorithm correct does not mean its implementation on a real system will necessarily always behave correctly.

Summary

- In this chapter we have discussed a wide range of problems that can occur in distributed systems.
  - Network delay;
  - Node out of sync (even with NTP)
  - Pause of execution (maybe due to GC)
- Such **partial failures** can occur is the defining characteristic of distributed systems.
- In distributed systems, we try to build tolerance of partial failures into software, so that the system as a whole may continue functioning even when some of its constituent parts are broken.
- To tolerate faults, the first step is to detect them, but even that is hard.
- Once a fault is detected, making a system tolerate it is not easy either: there is no global variable, no shared memory, no common knowledge or any other kind of shared state between the machines. (e.g. Solution: Quorum to agree)
- If you can avoid opening Pandora's box and simply keep things on a single machine, it is generally worth doing so.

<Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems>

PREVIOUS POST
Designing Data Intense Application – Chapter 7:Transactions

NEXT POST
Designing Data Intense Application – Chapter 9: Consistency and Consensus